Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Kurzdokumentation zum D·CORE-Assembler und Emulator

1 Die D·CORE-Assemblersprache

Die folgenden Ausführungen setzen die Kenntnis des Befehlssatzes des D·CORE-Prozessors voraus. Hierfür sei insbesondere auf die Aufgabenblätter des RSB-Praktikums verwiesen, siehe `https://tams.informatik.uni-hamburg.de/lectures/` im Wintersemester.

Den hier vorgestellten Assembler mit integriertem Emulator gibt es als ausführbares Binary für Linux, Windows und für MAC. Das Programm kann von der Webseite des Praktikums heruntergeladen werden.

- `t3asm` Zip-Archiv des Binary für Linux
- `t3asm.exe` Zip-Archiv des Binary für Windows
- `macT3asm` komprimiertes Disk Image für MAC-OS; starten mit ctrl+click oder Gatekeeper „bypass-dialog“

Der Assembler/Emulator ermöglicht, ein Assembler-Programm zu editieren, zu assemblieren und im integrierten Emulator zu testen. Der erzeugte Maschinencode kann auch in den Speicher des HADES-Modells geladen und hier simuliert werden. Unterstützt wird der gesamte in der Praktikumsdokumentation angegebene Befehlssatz und einige zusätzliche Pseudobefehle, auf die später eingegangen wird.

Bitte beachten Sie folgende Eigenschaften:

- In der Eingabedatei sind die Assemblerbefehle zeilenweise organisiert, in Regel mit einem Befehl pro Zeile. Häufig werden Label, wegen der besseren Übersichtlichkeit, als extra Zeile geschrieben, wie in den Beispielen.
- Der Assembler unterscheidet nicht zwischen Groß- und Kleinschreibung.
- Leerzeichen werden vom Parser überlesen und können bei eindeutiger Syntax auch entfallen.
- Kommentare beginnen mit einem ‘`;`’ und enden am Ende der Zeile.
- Ein symbolisches Label ist eine Zeichenkette, die mit einem Buchstaben beginnt und mit einem Doppelpunkt abgeschlossen wird.
- Zahlen können sowohl dezimal als auch hexadezimal eingegeben werden. Hex.-Zahlen werden mit dem Präfix `0x` oder einem `$` Zeichen gekennzeichnet, z.B. `$FFFF`, `0xFFF`, `$10`, `0x0`.
- Die sechzehn Register des D·CORE tragen die Namen `R0`, . . . , `R15`.
- Die Reihenfolge der Register auf Assemblerebene kann anders als im Binärcode der D·CORE-Befehle sein. So wird beispielsweise bei den ALU-Befehlen das Zielregister `Rx` zuerst genannt. Siehe dazu Tabelle 1 auf der folgenden Seite.

Mnemonic	Codierung	Hex	Bedeutung
			ALU-Operationen
mov	0010 0000 ⟨yyyy⟩ ⟨xxxx⟩	20yx	R[x] = R[y]
addu	0010 0001 ⟨yyyy⟩ ⟨xxxx⟩	21yx	R[x] = R[x] + R[y]
addc	0010 0010 ⟨yyyy⟩ ⟨xxxx⟩	22yx	R[x] = R[x] + R[y] + C (updates C)
subu	0010 0011 ⟨yyyy⟩ ⟨xxxx⟩	23yx	R[x] = R[x] - R[y]
and	0010 0100 ⟨yyyy⟩ ⟨xxxx⟩	24yx	R[x] = R[x] AND R[y] (bitwise)
or	0010 0101 ⟨yyyy⟩ ⟨xxxx⟩	25yx	R[x] = R[x] OR R[y] (bitwise)
xor	0010 0110 ⟨yyyy⟩ ⟨xxxx⟩	26yx	R[x] = R[x] XOR R[y] (bitwise)
not	0010 0111 ⟨****⟩ ⟨xxxx⟩	27*x	R[x] = NOT R[x] (bitwise)
			Shift-Operationen
lsl	0010 1000 ⟨yyyy⟩ ⟨xxxx⟩	28yx	R[x] = R[x] << R[y].<3:0>
lsr	0010 1001 ⟨yyyy⟩ ⟨xxxx⟩	29yx	R[x] = R[x] >>> R[y].<3:0>
asr	0010 1010 ⟨yyyy⟩ ⟨xxxx⟩	2Ayx	R[x] = R[x] >> R[y].<3:0>
lslc	0010 1100 ⟨****⟩ ⟨xxxx⟩	2C*x	R[x] = R[x] << 1, C=R[X].15
lsrc	0010 1101 ⟨****⟩ ⟨xxxx⟩	2D*x	R[x] = R[x] >>> 1, C=R[X].0
asrc	0010 1110 ⟨****⟩ ⟨xxxx⟩	2E*x	R[x] = R[x] >> 1, C=R[X].0
			Vergleichs-Operationen
cmpe	0011 0000 ⟨yyyy⟩ ⟨xxxx⟩	30yx	C = (R[x] == R[y])
cmpne	0011 0001 ⟨yyyy⟩ ⟨xxxx⟩	31yx	C = (R[x] != R[y])
cmpgt	0011 0010 ⟨yyyy⟩ ⟨xxxx⟩	32yx	C = (R[x] > R[y]) (signed)
cmplt	0011 0011 ⟨yyyy⟩ ⟨xxxx⟩	33yx	C = (R[x] < R[y]) (signed)
			Immediate-Operationen
movi	0011 0100 ⟨cccc⟩ ⟨xxxx⟩	34cx	R[x] = 0x000⟨c⟩
addi	0011 0101 ⟨cccc⟩ ⟨xxxx⟩	35cx	R[x] = R[x] + 0x000⟨c⟩
subi	0011 0110 ⟨cccc⟩ ⟨xxxx⟩	36cx	R[x] = R[x] - 0x000⟨c⟩
andi	0011 0111 ⟨cccc⟩ ⟨xxxx⟩	37cx	R[x] = R[x] AND 0x000⟨c⟩
lsli	0011 1000 ⟨cccc⟩ ⟨xxxx⟩	38cx	R[x] = R[x] << ⟨c⟩
lsri	0011 1001 ⟨cccc⟩ ⟨xxxx⟩	39cx	R[x] = R[x] >>> ⟨c⟩
bseti	0011 1010 ⟨cccc⟩ ⟨xxxx⟩	3Acx	R[x] = R[x] | (1 << ⟨c⟩) (set bit)
bclri	0011 1011 ⟨cccc⟩ ⟨xxxx⟩	3Bcx	R[x] = R[x] & !(1 << ⟨c⟩) (clear bit)
			Speicher-Operationen
ldw	0100 ⟨cccc⟩ ⟨yyyy⟩ ⟨xxxx⟩	4cyx	R[x] = MEM(R[y] + (0x000⟨c⟩<<1))
stw	0101 ⟨cccc ⟨yyyy⟩ ⟨xxxx⟩	5cyx	MEM(R[y] + (0x000⟨c⟩<<1)) = R[x]
			Kontrollfluss / Sprung-Operationen
br	1000 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	8iii	PC = PC+2+⟨*imm12*⟩
jsr	1001 ⟨iiii ⟨iiii⟩ ⟨iiii⟩	9iii	R[15] = PC+2; PC = PC+2+⟨*imm12*⟩ (call)
bt	1010 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	Aiii	(C=1) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
bf	1011 ⟨iiii ⟨iiii⟩ ⟨iiii⟩	Biii	(C=0) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
jmp	1100 ⟨****⟩ ⟨****⟩ ⟨xxxx⟩	C**x	PC = R[x]
halt	1111 ⟨****⟩ ⟨****⟩ ⟨****⟩	F***	halt
andere Opcodes illegal			

⟨xxxx⟩, x: 4-bit Index des Quell- und Zielregisters RX ⟨xxxx⟩ – binär, x – hex
⟨yyyy⟩, y: 4-bit Index des Quellregisters RY ⟨yyyy⟩ – binär, y – hex
⟨cccc⟩, c: 4-bit Konstante IMM4 ⟨cccc⟩ – binär, c – hex
iii: 12-bit sign-extended Konstante IMM12 ⟨iiii⟩ – binär, i – hex
⟨****⟩, *: don't care

Tabelle 1: D·CORE Befehle – Binärcode

1.1 Assembler-Befehle

Nachfolgend wird die Syntax der D·CORE-Assemblerbefehle beschrieben, die Semantik sollte aus den ersten Praktikumsaufgaben bekannt sein. Sie ist in Tabelle 1 skizziert. Außer den bereits beschriebenen Assemblerbefehlen, die die Maschinenbefehle des D·CORE-Prozessors abbilden, werden normalerweise noch weitere Befehle, so genannte Pseudo-Befehle, für ein sinnvolles Assembler-Programm benötigt.

1.1.1 ALU-Operationen

Befehle: `mov`, `addu`, `addc`, `subu`, `and`, `or`, `xor` und `not`

Syntax

```
[⟨Label⟩:]  ⟨asmOp⟩ ⟨Rx⟩, ⟨Ry⟩  [; ⟨Kommentar⟩]

⟨asmOp⟩ ::= mov|addu|addc|subu|and|or|xor|not
```

Beispiel

```
    mov   R0, R1          ; Inhalt von R1 nach R0 kopieren: R0=R1
    addu  R0, r2          ; R2 zu R0 addieren:              R0=R0+R2
```

1.1.2 Schiebe-Operationen

Befehle: `lsl`, `lsr`, `asr`, `lslc`, `lsrc` und `asrc`

Syntax

```
[⟨Label⟩:]  ⟨asmOp⟩ ⟨Rx⟩, ⟨Ry⟩  [; ⟨Kommentar⟩]

⟨asmOp⟩ ::= lsl|lsr|asr|lslc|lsrc|asrc
```

Beispiel

```
    lsl   R0, R1          ; R0 um R1(3..0) Bit nach links schieben
    asrc  R2, R1          ; arith. Rechtsschieben von R2 um 1 Bit; C=R2(0)
```

1.1.3 Vergleichs-Operationen

Befehle: `cmpe`, `cmpne`, `cmpgt` und `cmplt`

Syntax

```
[⟨Label⟩:]  ⟨asmOp⟩ ⟨Rx⟩, ⟨Ry⟩  [; ⟨Kommentar⟩]

⟨asmOp⟩ ::= cmpe|cmpne|cmpgt|cmplt
```

Beispiel

```
    cmpe  R0, R1          ; C=1 wenn (R0 == R1)
    cmplt R2, R1          ; C=1 wenn (R2 <  R1)
```

1.1.4 Immediate-Operationen

Befehle: `movi`, `addi`, `subi`, `andi`, `lsli`, `lsri`, `bseti` und `bclri`

Syntax

```
[⟨Label⟩:]  ⟨asmOp⟩ ⟨Rx⟩, ⟨Ry⟩  [;⟨Kommentar⟩]

⟨asmOp⟩ ::= movi|addi|subi|andi|lsli|lsri|bseti|bclri
```

Beispiel

```
    movi  R10, 0          ; Inhalt von R10 auf 0 setzen\\
    addi  R10, $F         ; 15 auf R10 aufaddieren: R10=R10+15\\
```

1.1.5 Speicher-Operationen

Befehle: `ldw` und `stw`

Bitte beachten Sie, dass beide Befehle die gleiche Syntax haben. Das erste Argument bezeichnet das Ziel-/Quellregister, das Zweite die Speicheradresse mit Offset. Damit ist bei `ldw` das Ziel links, bei `stw` die Quelle. Der Offset der Speicheradresse wird in ⟨*Bytes*⟩ gezählt. Erlaubt sind also Werte im Bereich von `0...31`, wobei ungerade Werte keinen Sinn ergeben, aber nicht als Fehler betrachtet werden. Ein Offset von 0 kann auch entfallen. Negative Offsets sind nicht möglich.

Syntax

```
[⟨Label⟩:]  ⟨asmOp⟩ ⟨Rx⟩, [⟨offset⟩](⟨Ry⟩)  [;⟨Kommentar⟩]

⟨asmOp⟩  ::= ldw|stw
⟨offset⟩ ::= 0...31
```

Beispiel

```
    ldw   R10, (R2)       ; Lade R10 mit dem Wert, dessen Adresse in R2 steht
    stw   R10, 2(R2)      ; Speichere den Inhalt von R10 unter der Adresse R2+2
```

1.1.6 Kontrollfluss / Sprung-Operationen

Befehle: `br`, `jsr`, `bt`, `bf` und `jmp`

Syntax

```
[⟨Label⟩:]  ⟨asmOp⟩   [; ⟨Kommentar⟩]
[⟨Label⟩:]  jmp ⟨Rx⟩  [; ⟨Kommentar⟩]

⟨asmOp⟩ ::= br|jsr|bt|bf
```

Beispiel

```
    ldw   R10, (R2)       ; Lade R10 mit dem Wert, dessen Adresse in R2 steht
    movi  R0, 0
LOOP:
    jsr   SUBPROG         ; Unterprogrammaufruf SUBPROG
    subi  R8, 1           ; R8 = R8-1
    cmpe  R8, R0          ; Vergleich R8 == 0
    bf    loop            ; Sprung auf LOOP, wenn R8 != 0
    ...
SUBPROG:
    ...
    jmp   R15             ; Rücksprung aus Unterprogramm
                          ; Konvention: jsr speichert Rücksprungadresse in R15
```

1.1.7 sonstige Operationen

Befehle: `halt`, `rfi` und `eepc`

Zum Testen der eigenen Programme sollte jedes Programm als letzten Befehl einen HALT-Befehl haben. Die beiden anderen Befehle `rfi` (Return From Interrupt) und `eepc` (Exchange EPC) dienen der Interruptbehandlung. Sie werden in den aktuellen Aufgaben nicht benutzt.

Syntax

```
[⟨Label⟩:]  ⟨asmOp⟩  [; ⟨Kommentar⟩]

⟨asmOp⟩ ::= halt|rfi|eepc
```

Beispiel

```
    ...
    halt                ; Programm anhalten (Endlosschleife im Mikroprogramm)
```

1.1.8 Trap-Befehl

Befehl: `trap`

Dieser Befehl wird vom HADES-Modell noch nicht unterstützt, wohl aber vom Assembler, wenn die entsprechende Option gewählt wird. Weiter unten wird genauer darauf eingegangen.

Syntax

```
[⟨Label⟩:]  trap ⟨Offset⟩  [; ⟨Kommentar⟩]
```

1.2 Pseudo-Befehle

Außer den bereits beschriebenen Assemblerbefehlen, die die Maschinenbefehle des D·CORE-Prozessors abbilden, werden normalerweise noch weitere Befehle, so genannte Pseudo-Befehle, für ein sinnvolles Assembler-Programm benötigt. Die Pseudo-Befehle beginnen alle mit einem Punkt `.`⟨*psdOp*⟩, um sie klar von den Maschinenbefehlen des Prozessors abzusetzen.

1.2.1 .org

Manchmal ist es nötig, genau festzulegen, wo ein bestimmter Teil des Programms im Speicher abgelegt wird. Beim D·CORE muss z.B. die Service-Routine eines Interrupts auf einer festgelegten, vordefinierte Adresse liegen (`$100`). Dasselbe Problem tritt auf, wenn Daten vom Assembler im RAM ablegt werden sollen. Dazu dient der `.org`-Befehl. Dieser (Pseudo-) Befehl setzt den Adresszähler des Assemblers auf den angegebenen Wert; die Wirkung dieses Befehls ist im Maschinenprogramm daran zu erkennen, dass die auf `.org` folgenden Maschinenbefehle ab der angegebenen Adresse im Speicher liegen. Per Default ist die Anfangsadresse des Assemblers, wie auch die Startadresse des `pc`, auf `0x0` eingestellt.

Syntax

```
[⟨Label⟩:]  .org ⟨Adresse⟩  [; ⟨Kommentar⟩]
```

Beispiel

```
    .org  0              ; Adresszähler auf 0 setzen (nicht nötig, s.o.)
    movi  R0, 0          ; Befehl steht auf Adresse: 0x0000
    bseti R0, 15         ; Befehl steht auf Adresse: 0x0002
    jmp   R0             ; Befehl steht auf Adresse: 0x0004
    .org  $8000          ; Adresszähler auf 0x8000 setzen
    movi  R1, 1          ; Befehl steht auf Adresse: 0x8000
```

1.2.2 .defw (define word)

Dieser Befehl dient dazu, ein Speicherwort mit einem bestimmten Wert zu belegen. Die Adresse des Speicherwortes wird durch den aktuellen Wert des Adresszählers des Assemblers (s.o.) bestimmt.

Syntax

```
[⟨Label⟩:]  .defw ⟨Wert⟩  [; ⟨Kommentar⟩]
```

Beispiel

```
    .defw $FFFF          ; Legt den Wert -1 auf der aktuellen Adresse ab
    .org  $8000          ; Adresszähler auf 0x8000 setzen
    .defw $7000          ; Legt 0x7000 im Speicher auf Adresse 0x8000 ab
```

1.2.3 .ascii, .assho

Diese (identischen) Befehle dienen dazu, einen String wortweise im Speicher abzulegen. Man beachte, dass wirklich nur der String abgelegt wird, aber nicht das terminierende Null-Wort, wie es z.B. bei null-terminierten Strings in der Programmiersprache C erforderlich ist.

Syntax

```
[⟨Label⟩:]  .ascii "⟨String⟩"  [; ⟨Kommentar⟩]
```

Beispiel

```
    .ascii "Hallo"       ; Legt den String "Hallo" ab der aktuellen Adresse
                         ; des Adresszählers im Speicher ab
```

1.2.4 .defs (define storage)

Dieser Befehl dient dazu, Speicherplatz, z.B. für ein Array, zu reservieren. Der Inhalt der reservierten Speicherzellen ist (natürlich) undefiniert.

Syntax

```
[⟨Label⟩:]  .defs ⟨Words⟩  [; ⟨Kommentar⟩]

 ⟨Words⟩  ::=  Anzahl der Speicherworte
```

Beispiel

```
      .org   $200
NUL:  .defw  0             ; 0x0000 in Speicherwort an Adresse     0x0200
ARR:  .defs  4             ; Reserviert 4 Speicherworte; Adressen: 0x0202..0x0208
STR:  .ascii "??"          ; String "??" ab der Adresse            0x020A
```

1.2.5 .equ (equate)

Dieser Befehl dient dazu, einen konstanten Wert (assemblerintern) zu definieren. Es wird also kein Speicherplatz belegt.

Syntax

```
[⟨Label⟩:]  .equ ⟨Zahl⟩  [; ⟨Kommentar⟩]
```

Beispiel

```
RAM:  .equ   $8000       ; Weist RAM den Wert 0x8000 zu
      .org   RAM         ; Adresse setzen
      .defw  RAM         ; Inhalt einer Speicherzelle setzen

      br     RAM         ; FEHLER:  Sprung zu... (0x0000!)
                         ; ACHTUNG: Branch-Offset ist nur 12 Bit breit
```

1.2.6 .end

Dieser Befehl dient dazu, das Assemblieren zu beenden. In der Regel fehlt der Befehl im Assemblercode gänzlich.

Syntax

```
[⟨Label⟩:]  .end  [; ⟨Kommentar⟩]
```

Beispiel

```
      br     GOON
      .end               ; FEHLER: nach .end wird nicht mehr assembliert
GOON:                    ; das Label GOON ist nicht mehr in der Symboltabelle
      ...
```

1.2.7 .stack, .push, .pop

`.stack` legt das Register fest, das als Stackpointer verwendet wird und welches dann von den Pseudobefehlen (Makros) `.push` und `.pop` implizit verwendet wird. Das Default-Register ist `R0`. Es ist die Aufgabe des Programmierers, den Stackpointer mit einem Wert zu initialisieren, der im Adressbereich des RAMs liegt. Die Adresse referenziert immer das zuletzt benutzte Element des Stacks. Der `.stack`-Befehl erzeugt keinen ausführbaren Code.

Syntax

```
[⟨Label⟩:]  .stack ⟨Rx⟩  [; ⟨Kommentar⟩]
[⟨Label⟩:]  .push  ⟨Ry⟩  [; ⟨Kommentar⟩]
[⟨Label⟩:]  .pop   ⟨Ry⟩  [; ⟨Kommentar⟩]

 ⟨Rx⟩  ::=  Stackpointer
 ⟨Ry⟩  ::=  Quell-/Zielregister

 Codeäquivalente:
 .push ⟨Ry⟩   ::=  subi ⟨Rx⟩, 2
                   stw  ⟨Ry⟩, (⟨Rx⟩)
 .pop  ⟨Ry⟩   ::=  ldw  ⟨Ry⟩, (⟨Rx⟩)
                   addi ⟨Rx⟩, 2
```

Beispiel

```
    .stack R14          ; R14 als Stackpointer verwenden
    movi   R14, 0       ; Initialisieren  R14=0x8080
    bseti  R14, 15      ;
    bseti  R14, 7       ;
                        ;                                   R14=      Stack
    .push  R1           ; R1 auf Stack sichern (Caller-Save) 0x807E   <R1>
    .push  R2           ; R2 auf Stack sichern               0x807C   <R2>
    jsr    SUBPRG       ; Unterprogrammaufruf
    .pop   R2           ; R2 zurücksichern                   0x807E   <R1>
    .pop   R1           ; R1 zurücksichern                   0x8080    --
```

1.2.8 .prdez, .prstr, .prnewline

Diese drei Pseudo-Befehle sind für den Emulator, um dort im Displayfenster Zahlen und Strings auszugeben, siehe auch Abschnitt 3.1.3.

Syntax

```
[⟨Label⟩:]  .prdez ⟨Rx⟩   [; ⟨Kommentar⟩]
[⟨Label⟩:]  .prstr ⟨Ry⟩   [; ⟨Kommentar⟩]
[⟨Label⟩:]  .prnewline    [; ⟨Kommentar⟩]

 ⟨Rx⟩  ::=  wird als Dezimalzahl ausgegeben
 ⟨Ry⟩  ::=  Startadresse des null-terminierten Strings
```

1.2.9 Beispiel

Zum Abschluss dieses Teils hier noch ein Beispielprogramm, das die Zahlen in zwei Feldern addiert und das Ergebnis in einem dritten Feld ablegt:

Beispiel

```
    br    START
    .defw FELD1          ; Adresse Array 1 (Op1)
    .defw FELD2          ; Adresse Array 2 (Op2)
    .defw FELD3          ; Adresse Array 3 (Summe = Op1+Op2)
START:
    movi  R4,  0         ; Basis-Adresse
    ldw   R10, 2(R4)     ; Adresse Summand 1
    ldw   R11, 4(R4)     ; Adresse Summand 2
    ldw   R12, 6(R4)     ; Adresse Summe
    movi  R2,  5         ; Schleifenzähler
LOOP:
    ldw   R0, (R10)      ; lade Summand 1
    ldw   R1, (R11)      ; lade Summand 2
    addu  R1, R0         ; berechne  Summe
    stw   R1, (R12)      ; speichere Summe
    subi  R2, 1          ; dekrementiere Schleifenzähler
    cmpe  R2, R4         ; prüfe Abbruchbedingung
    bt    ENDE           ; (C==1): Ende
    addi  R10, 2         ; erhöhe Adresse für Summand 1
    addi  R11, 2         ; erhöhe Adresse für Summand 2
    addi  R12, 2         ; erhöhe Adresse für Summe
    br    LOOP
ENDE:
    halt                 ; Programmende
;=============================================================================
    .org  $8000          ; RAM Startadresse
FELD3:                   ; 5 Worte Ergebnis: 0x8000..0x8008
    .defs 5
FELD2:                   ; 5 Worte Operand2: 0x800A..0x8012
    .defw 0
    .defw 10
    .defw 20
    .defw 30
    .defw 40
FELD1:                   ; 5 Worte Operand1: 0x8014..0x801C
    .defw 40
    .defw 30
    .defw 20
    .defw 10
    .defw 0
    .end
```

2 Der Assembler

Das Hauptfenster des D·CORE-Assemblers ist denkbar einfach: eine Menü-Leiste mit den üblichen Datei- und Edit-Einträgen und einem Werkzeug- und Hilfebutton. Darunter eine Schnellzugriffsleiste für häufig benutzte Funktionen, wie Datei öffnen bzw. neu anlegen, über sichern bis zu assemblieren und schließlich Starten des Emulators. Dominiert wird das Hauptfenster durch das Editorfenster, zum Bearbeiten bzw. Erstellen des Assemblerprogramms (siehe Abb: 1). Wird das Assemblieren des Quelltextes ohne Fehlermeldung abgeschlossen kann der Emulator gestartet werden. Sollten während der Assemblierung Fehler auftreten, werden diese in der Statusleiste angezeigt und die verantwortliche Programmzeile zum Korrigieren farblich hervorgehoben.

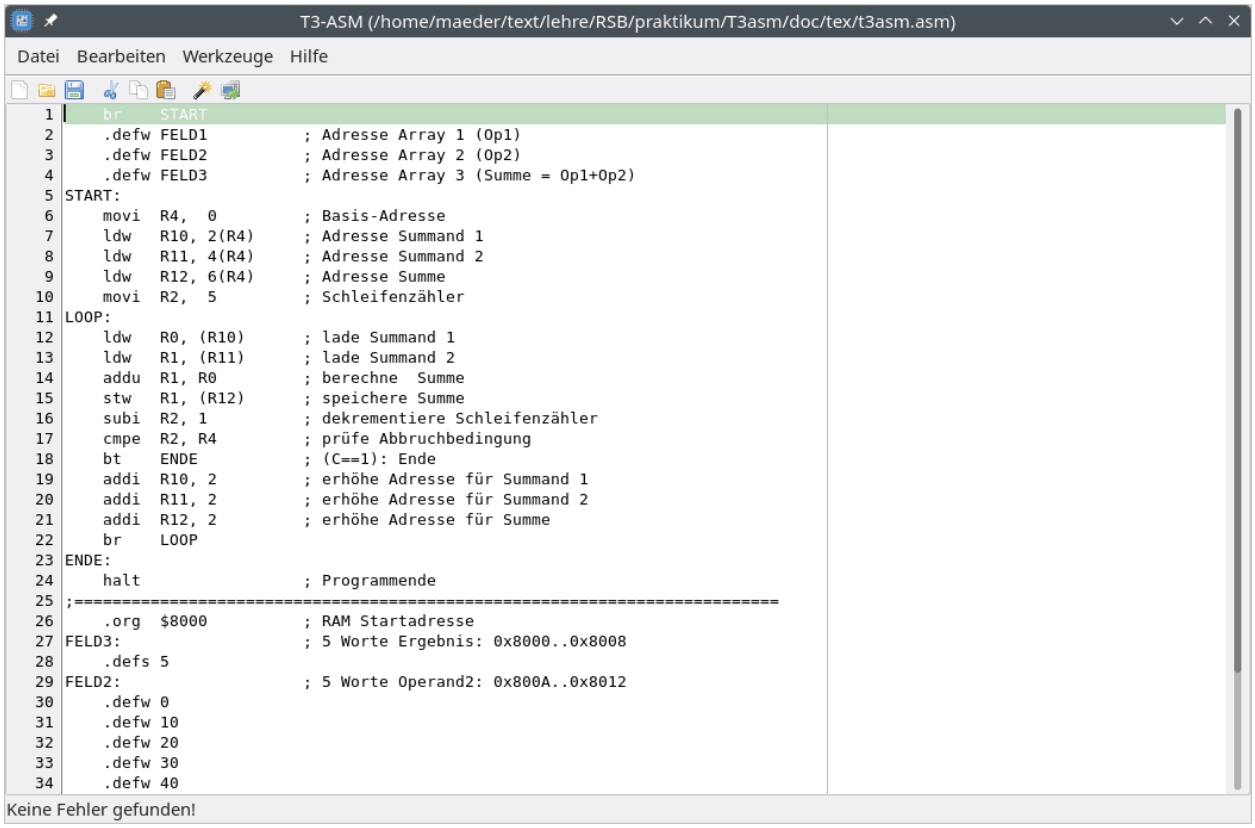

Abbildung 1: Fenster des D·CORE-Assemblers

2.1 Optionen der Assemblierung bzw. des Emulators

Über den Menüpunkt *Optionen* (*Bearbeiten* → *Optionen*) lassen sich verschiedene Parameter auswählen:

2.1.1 RAM/ROM-Files

Es werden zwei Dateien erzeugt, die in das ROM bzw. das RAM des HADES-Modells vom D·CORE-Prozessor geladen werden können: ⟨*name*⟩`.rom` und ⟨*name*⟩`.ram`.

2.1.2 Listing

Es wird ein vollständiges Listing des Programms mit Zeilennummer, Adresse und Opcode erzeugt: ⟨*name*⟩`.lst`.

2.1.3 Altera

Es wird eine ROM-Datei (⟨*name*⟩`.cod`) für den Altera RAM/ROM-Generator erzeugt. Dieses Format ist veraltet.

2.1.4 Byte-Offset

Die Byteadressierung wird standardmäßig verwendet und sollte angeschaltet bleiben.

2.1.5 Steps?

Bei aktivierter Steps?-Option wird die Anzahl der nach der Betätigung der Schaltfläche `GO` ausgeführten Instruktionen angezeigt. Diese Option wird nicht mehr unterstützt.

2.1.6 $-Style

Legt fest, wie der Emulator Hexadezimalzahlen anzeigen soll. Default ist die Form **`0x????`**. Nach Auswahl dieser Option werden die Zahlen in der Form **`$????`** angezeigt. Dieses betrifft nicht die Eingabe; hier sind immer beide Formen möglich.

2.1.7 Trap?

Nach Auswahl kann der Trap-Befehl verwendet werden. Dabei ist zu beachten, dass das HADES-Modell diese Anweisung nicht unterstützt. Weitere Einzelheiten sind unten in den Ausführungen zur ISR-Adresse zu finden.

2.1.8 SimuRAM

Ohne Funktion

2.1.9 MMU

Das D·CORE-System verfügt über eine einfache Speicherverwaltungseinheit (MMU: *M*emory *M*anagement *U*nit). Um sie zu nutzen, ist diese Option zu aktivieren. Folgende Festlegungen wurden getroffen, damit die Speicherverwaltung auf ein Minimum zu reduziert werden kann:

- Seitengröße: 8 KiB
- keine Statusinformationen zu den Seiten

Daraus ergibt sich bei einem 16bit-Adressraum des D·CORE und einer Speicherwortbreite von ebenfalls 16bit (2 Byte) eine Seitentabelle mit $\frac{2 \cdot 2^{16} Bytes}{8 \cdot 2^{10} Bytes} = \frac{2^{17}}{2^{13}} = 2^4$ Einträgen. Zur Umsetzung der virtuellen Adressen in physikalische Adressen genügt daher ein einstufiges Adressumsetzungsschema (siehe Abb. 2) und damit lediglich ein kleines RAM mit 16 Worten für die Basisadresse der Seite (die obersten vier Adressbits der virtuellen Adresse), während die unteren 12 Bit der der virtuellen Adresse als Seitenoffset direkt an die physikalische Adresse durchgereicht werden.

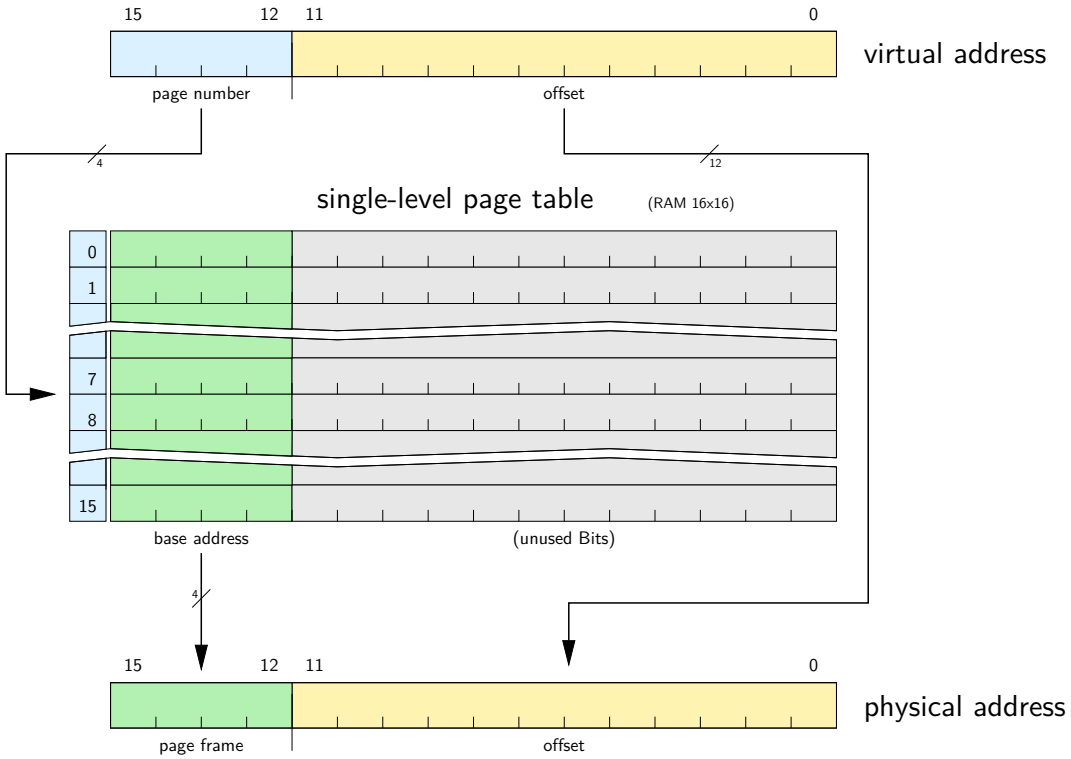

Abbildung 2: Einfache einstufige Adressumsetzung ohne Statusbits. Da aus Gründen der Einheitlichkeit anstelle eines 16x4 Bit RAM ein 16x16 Bit RAM verwendet wurde, sind die niederwertigen 12 Bit der Seitentabelle „don't care".

Die Seitentabelle wird unter den Adressen `0x7010` .. `0x702E` in den Adressraum eingeblendet. Die Abbildung 3 zeigt mögliche Belegungen der Seitentabelle:

links: die Seitentabelle nach Reset mit Abbildung des nahezu gesamten Adressraumes auf die erste Seite des ROMs, wobei lediglich der Index 7 als Page Table Entry (PTE) die `0111` enthält. In diesem Adressbereich liegen beispielsweise die Memory Mapped I/O Adressen und die Seitentabelle selbst (deshalb ist dieser Eintrag auch zwingend erforderlich).

mitte: hier ist ein lineares Mapping illustriert.

rechts: diese Abbildung zeigt ein Mapping, wie es beim D·CORE mit einem Prozess und Zugriff auf ROM und RAM möglich wäre. Es kann hier letztendlich neben dem eingeblendeten Adressbereich nur auf zwei weitere physikalische Seiten (erste Seite im ROM und erste Seite im RAM) zugegriffen werden.

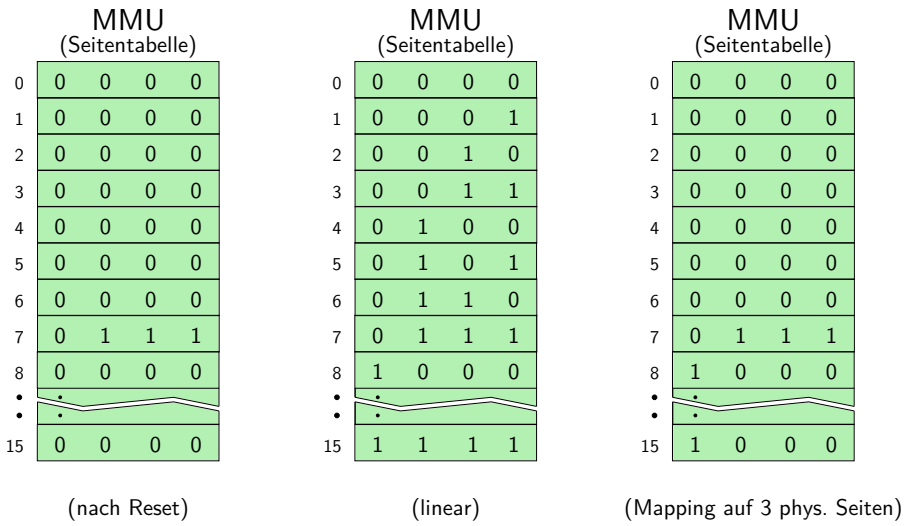

Abbildung 3: Beispiele für die Adressumsetzung (es werden jeweils nur die vier relevanten Bits der Seitentabelle (siehe Abb: 2) gezeigt)

2.1.10 Interr.?

Diese Option schaltet die Interruptbehandlung ein. Im Emulator kann durch die Betätigung der Schaltfläche *Interrupt* ein Interrupt ausgelöst werden. Nach Abarbeitung des aktuellen Befehls wird die, beginnend an der Adresse `0x0100`, abgelegte Interrupt Service-Routine angesprungen und nach deren Abarbeitung wieder in das unterbrochene Programm zurück gesprungen (siehe ISR-Adresse).

2.1.11 ISR-Adresse

Gibt die Adresse der Interrupt Service-Routine (TRAP nicht aktiviert), bzw. der Interrupt-Vektor-Tabelle (TRAP aktiviert) an. Voreingestellt ist die Adresse `0x0100`, wie im HADES-Modell. Im Folgenden soll kurz der Unterschied erläutert werden, wobei das HADES-Modell nur die Interrupt Service-Routine unterstützt. Die ISR-Adresse ist dabei die Adresse, unter der die Routine steht, die ausgeführt wird, wenn ein Interrupt erfolgt. Beispielsweise könnte es wie folgt aussehen:

Beispiel

```
    jsr   INPUT
    movi  R0, 0
WAIT:                   ; Wartet auf einen Interrupt
    br    WAIT
INPUT:
    . . .
    jmp   R15           ; Rücksprung
;===============================================================================
    .org  $100          ; ISR-Adresse
    addi  R0, 1         ; Wenn Interrupt, dann R0=R0+1
    jsr   OUTPUT        ; R0 ausgeben oder etwas anderes
    rfi                 ; Rücksprung
```

Wurde die Trap?-Option aktiviert, dann ist die ISR-Adresse ein Zeiger auf eine Liste von Unterprogramm-Adressen, die mit Hilfe des Trap-Befehls angesprungen werden können. Insbesondere liegt auf der ISR-Adresse die Adresse der normalen Interrupt Service-Routine. Obiges Beispiel könnte man dann auch wie folgt codieren:

Beispiel

```
    trap  1
    movi  R0, 0
WAIT:                   ; Wartet auf einen Interrupt
    br    WAIT
INPUT:
    . . .
    rfi                 ; Rücksprung
;=======================================================================
    .org  $100
    .defw ISR           ; zeigt auf Routine, die den Interrupt behandelt
    .defw INPUT         ; normale Eingabe
;=======================================================================
    .org  $200
ISR:
    addi  R0, 1         ; Wenn Interrupt, dann R0=R0+1
    jsr   OUTPUT        ; R0 ausgeben oder etwas anderes
    rfi                 ; Rücksprung
```

Es stellt sich natürlich die Frage, welchen Vorteil dieses Konzept mit sich bringt. Ein Vorteil für den D·CORE-Prozessor ist, dass die Routine, die angesprungen werden soll, irgendwo im Speicher liegen kann und nicht nur innerhalb eines 12 Bit-Offsets zum aktuellen Wert des Programmzählers wie beim JSR-Befehl. Eine weiterer Vorteil liegt viel allgemeiner eher auf der Betriebssystemebene, wo es Basisroutinen geben muss, die von allen Anwendungsprogrammen genutzt werden, beispielsweise die Ausgabe eines Zeichens auf den Bildschirm oder das Lesen von der Tastatur. Bei einer neuen Betriebssystemversion ist dann zu befürchten, dass sich auch die Einsprungstellen dieser Unterprogramme ändern, was heißt, dass die Anwenderprogramme neu kompiliert, assembliert und gebunden werden müssen, was inakzeptabel wäre. Eine mögliche Lösung dieses Problems wäre, für diese Basisroutinen keine klassischen Unterprogramm-Aufrufe (beim D·CORE also `JSR`) zu verwenden, sondern Trap-Befehle, bei denen das Betriebssystem seine geänderten Einsprungstellen nur noch in einer modifizierten Tabelle abzulegen hat, was das Anwenderprogramm nicht berührt. Sollte jemand noch das etwas zweifelhafte Vergnügen gehabt haben Assembler-Programme für DOS zu schreiben, so kennt derjenige wahrscheinlich die `INT 24H` und `INT 10H` Aufrufe, die letztlich genau dieses Konzept realisieren.

3 Der Emulator

Nach erfolgreichem Assemblieren (s.o.) kann der Emulator aus dem Assembler-Fenster heraus gestartet werden: Schnellzugriff (*Monitor-Ikon*) oder im Menu (*Werkzeuge → Emulator Starten*). Es erscheint dann das in Abb: 4 gezeigte Fenster.

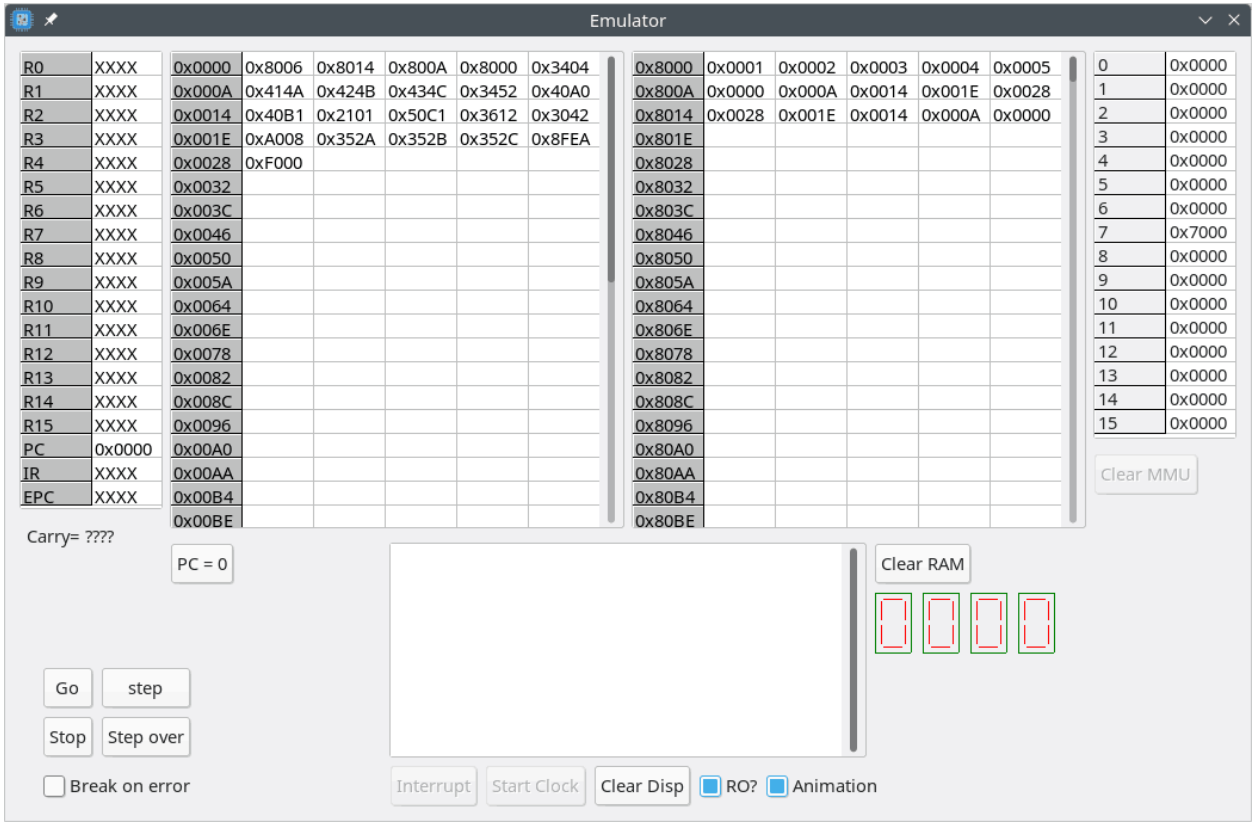

Abbildung 4: Fenster des D·CORE-Emulators

3.1 Anzeige- und Steuerelemente des Emulators

Der folgende Abschnitt erläutert die Anzeigeelemente mit den ggf. dazugehörigen Steuerelementen und die grundsätzlichen Schaltflächen zur Steuerung des Emulators (siehe Abb. 4).

3.1.1 Register, RAM und ROM des D·CORE

Am linken Rand nach unten auslaufend sind die 16 Register des D·CORE nebst PC, IR und EPC zu sehen. Rechts daneben wird das ROM angezeigt und daneben wiederum das RAM. Sowohl RAM und ROM sind jeweils rechts mit einem Slider versehen, um weitere Speicherzellen anzuzeigen. Sowohl Registerinhalte als auch Inhalte der Speicherzellen können bei pausierter Emulation verändert werden.

Über die Schaltfläche *PC=0* lässt sich der *PC* komfortabel auf `0x0` setzen und über *Clear RAM* das RAM löschen (alle Zellen auf `0x0000` setzen).

3.1.2 Die MMU

Am rechten Rand des Emulator-Fensters werden die Register der MMU (die Seitentabelle) angezeigt, die allerdings nur relevant sind, wenn die entsprechende Option (siehe 2.1.9) ausgewählt ist. Bei nicht gesetzter Option wird die Schaltfläche `Clear MMU` ausgegraut, mit der ggf. die Register der MMU, die Seitentabelle, gelöscht werden kann (auf `0x0000` setzen).

3.1.3 Das alphanumerische Display

Im unteren Bereich des Emulatorfensters befindet sich mittig ein Feld, das als Display dienen kann. Der Assembler kennt hierfür drei spezielle Pseudobefehle, um etwas auf diesem Display des Emulators ausgeben zu können (siehe auch Seite 8). Im RSB-Praktikum auf Bogen 4, Aufgabe 4.9 werden diese drei Befehle benutzt.

`.prdez` ⟨*reg*⟩ gibt den Inhalt des Registers ⟨*reg*⟩ als Dezimalzahl aus

`.prstr` ⟨*reg*⟩ gibt null-terminierten String aus, dessen Startadresse in Register ⟨*reg*⟩ steht

`.prnewline` gibt einen Zeilenumbruch aus

3.1.4 Sieben-Segment-Anzeige

Rechts neben dem Display befinden sich die vier Sieben-Segment-Anzeigen, die Memory Mapped über die Adresse *0x7002* angesprochen werden können. Bei einem auszugebenden 16 Bit Wert ergeben nur die Codierungen einer vierstelligen BCD-Zahl sinnvolle Anzeigen; andere Codes, beispielsweise für die Hex-Werte `A`...`F` erzeugen unsinnige Darstellungen.

3.1.5 Steuerungselemente unterhalb des Displays

Interrupt bei gesetzter *Interr.?*-Option (siehe Abschnitt 2.1.10) kann hier ein Interrupt ausgelöst werden.

Start Clock wenn die Option *Interr.?* gesetzt ist, können Interrupts nicht nur manuell (s.o.), sondern auch periodisch ausgelöst werden. Das Optionenfeld *Clock* legt die Anzahl der Takte zwischen Interrupts fest.

Clear Disp löscht das alphanumerische Display.

RO steht für read-only. Bei gesetztem *RO* ist das Terminal aus Sicht des Benutzers read-only, und somit aus Sicht des Prozessors write only, also ein Ausgabegerät. Bei nicht gesetztem *RO* kann das Terminal auch als Eingabegerät verwendet werden.[1]

Animation nach Abschalten dieser Option werden Animationen der GUI, wie beispielsweise das Markieren welche ROM-/RAM-Adressen gelesen/geschrieben werden, abgeschaltet. Dadurch läuft die Emulation geringfügig schneller.

[1] Da das in den aktuellen Aufgaben nicht benutzt wird, sind das dazu notwendige Adressmapping und die Benutzung hier nicht weiter beschrieben.

3.1.6 Steuerung des Emulators

Links neben dem Textdisplay befinden sich fünf Schaltflächen zur elementaren Steuerung des Emulationslaufs:

Go startet das Programm; die Emulation wird entweder durch *Stop* angehalten oder durch Ausführen eines Halt-Befehls beendet. Hat man vorher im Assembler-Fenster (siehe Seite 10) eine Zeile mit dem Cursor markiert, dann läuft der Emulator bis zu dieser Zeile. Dieses Verhalten ist vergleichbar mit dem „Breakpunkt“ eines Debuggers.

Stop stoppt die Programmausführung.

Step führt genau einen Befehl aus und färbt die entsprechende Zeile im Assemblercode ein.

Step over überspringt einen Unterprogramm-Aufruf und geht zum nachfolgenden Assemblerbefehl.

Break on error wird zur Laufzeit ein Fehler festgestellt, beispielsweise ein Schreibzugriff ins ROM, dann wird die Ausführung gestoppt und die auslösende Anweisung im Assemblercode hervorgehoben.